Deep Neural Network for Handwritten Digit Recognition

Jiaqi Zhang
AMATH 482 Winter 2017
University of Washington
zjq95@uw.edu

Abstract

Most people effortlessly recognize handwritten digits. This perceptive depends on human brains' series of visual cortices[1]. Images are first of all processed by the primary visual cortex V1, then passed on to a series cortices - V2, V3 until V5. These cortices are progressively more precise and accurate. Human brain can be thought of as a super powerful computer[1]. In this project, we want to show how to teach machine to recognize handwritten digits. The technique we use - Deep Neural Network, uses a cascade of multiple layers of units, each layer using the output from the previous layer to classify the handwritten digits into single digit integers.

I. Introduction and Overview

Deep Neural Networks (DNN) is a class of machine learning algorithm that uses a cascade of layers of units to extract features in order to classify the units into outcome classes. In this project, we use DNN to learn handwritten digit recognition. The dataset used is '*Mixed National Institute of Standards and Technology database'* (MNIST database)[2]. MNIST is a large database of handwritten digits, containing 60, 000 training images and 10, 000 testing images, each provided with its label. We use the training set to train our processing system and use the test set to measure the accuracy.

We will begin with a mathematical explanation of DNN, followed by showing how to implement the one layer DNN system to classify handwritten digits. Then we will utilize the Matlab toolbox 'Deep Neural Network' by Masayuki Tanaka[3] to proceed 2-layer, 3-layer recognition of the digits. Finally, we will discuss the computational results and observations.

II. Theoretical Background

2. 1 Perceptron

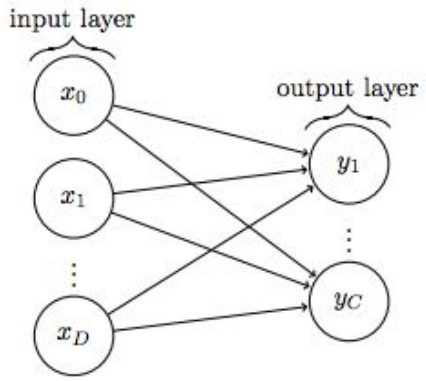

Figure 1. A perceptron containing D inputs and C outputs. The input units are propagated to an output using the weighted sum propagation rule [4].

Perceptrons have an input layer and an output layer. As shown in figure 1, each unit in the input layer X_i is mapped to exactly one unit in the output layer Y_j. Units in the input layer has a weight factor w_i, which indicates the bias of each unit.The propagation rule used in this project is weighted sum:

$$z_i = \sum_{k=1}^{D} w_{ik} x_k \tag{1}$$

Each output unit has a threshold that can be used to compare with the sum:

$$y_i = f(z_i) \tag{2}$$

In this case, a unit with a large weight *w* is likely to be mapped to the output unit with a large threshold *y.* This shows how bias is introduced.

2. 2 Multilayer Perceptron

Multilayer perceptron has hidden layers in between. Figure 2 shows a 2 layer perceptron. Its input layer is first mapped to the next hidden layer Y_k^1, then the result from the Y_k^1 is used as the input for the output layer Y_j. Multilayer perceptrons illustrates how different kinds of features are weighed in decision making. Each layer makes decisions by weighing up the results from the last layer, thus making the decision making more complex and sophisticated[2].

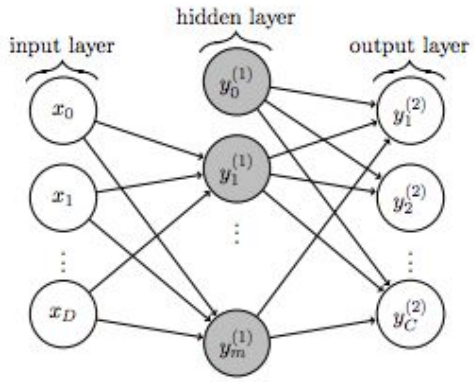

Figure 2. A 2-layer Perceptron with D inputs, m hidden layer units and C outputs[3].

2. 3 DNN on Handwritten Digit Classification

MNIST provides handwritten digits as images stored in row vectors as a matrix *S*, where S_{ij} represents the *jth* pixel of *ith* image. Figure 3 shows four random digit images from the MNIST dataset. Each image has *28x28* pixels, therefore, we have 784 input units for each image. Since we only work with single digits, we define 10 classes, in other words, 10 output neurons, each containing the activation value for 0, 1, 2, …, 9. Note that if the *jth* neuron contains the highest activation value, our program will evaluate this digits as *j-1.* The number of neurons used for the hidden layers are adjustable and we will experiment with different numbers of neurons for hidden layers as well as the number of hidden layers. These analysis can be found in the Computational Results section.

2. 4 Cross Validation

The goal for cross validation in DNN is to find the optimal set of neural network weights and bias values. Given the training set, we split it into two sets -- **training** and **validation** using a 5:1 proportion. In our case, the **tr** set has 50,000 images and the va set has 10,000 images. We use the **tr** set to train and apply the program on the **va** set to obtain error rate. Then we extract the set of weights and bias values that give the minimum error rate for later use. Then the set of parameters are used to train the original training set, which has 60,000 images.

Pseudocode for cross validation:

```
ite = 10
tr = subset of training(first 50000)
va = subset of training(last 10000)
for i  = 1: ite
     train tr, get param;
     Performance[i] = result from using paramon va.
end
get best performance and corresponding param.
use param to train training
```

III. Algorithm Implementation and Development

This section detaily presents the implementation of one layer DNN on handwritten digit recognition. Two-layer DNN is done by utilizing the DNN matlab toolbox. Parameters are changed and their effects will be discussed in the result section.

3.1 Loading Data

The original data contains the raw MNIST images stored in a binary format. The raw data is reshaped into a 28x28 matrix and converted to ‘double’ format, two helper functions ‘loadMNISTImages.m’ and ‘loadMNISTLabels.m’ are used. These files are provided by the Computer Science Department of Stanford University[5].

Figure 3. Sample digit images in MNIST dataset.

3. 2 Start Cross Validation

Cross validation is implemented to obtain the optimal sets of weights. This program is cross validated 20 times using a for loop which iterates 20 times.

3. 3 Data Preparation: Obtain *Tr, Va* Set, Label Matrix

First, the original training set is randomly permuted. Then the new training set is separated into the **tr** set and **va** set. Each containing 50,000 and 10,000 random rows of the original training set.

Labels of the training set is stored in a column vector of length 60,000. Each element represents the digit corresponding to the images. In order to obtain 10 units on the output layer, we reshape this column vector into a matrix *M* of dimension (10 * 60,000). M is initialized as a matrix full of 0s. If the *m*th label has a value *n*, then M_{nm} should be set to 1. Each column of *M* represents one label.

3. 4 Training Tr

Weight factors ***A*** are obtained by multiplying the ***Tr*** label matrix with the Moore-Penrose inverse of Tr. Moore-Penrose inverse is used because ***Tr*** is not a square matrix. This weight matrix ***A*** is multiplied to ***Va*** to get the classification result of the validation ***vResult***. Index of the maximum activation value in ***vResult*** is the digit label that we are looking for.

3. 5 Error Rate for Cross Validation

We subtract **Va**'s label matrix from **vResult**. The non-zero terms in the resulting matrix **vError** indicates difference, in other words, errors. Note that each difference will produce 2 non-zero numbers in the rows. Therefore when calculating error rate, we divide the number of mismatches by the total number of images, then divide by 2 to eliminate repetition. This is the end of the for loop for cross validation.

3.6 Choosing Optimal Weight to train 'trainSet' and test 'testSet'

Now we have the 20 error rates from cross validation and the corresponding weights ***Ai*** used. We extract the A that gives the minimum error rate and multiply it with the test set to obtain the result.

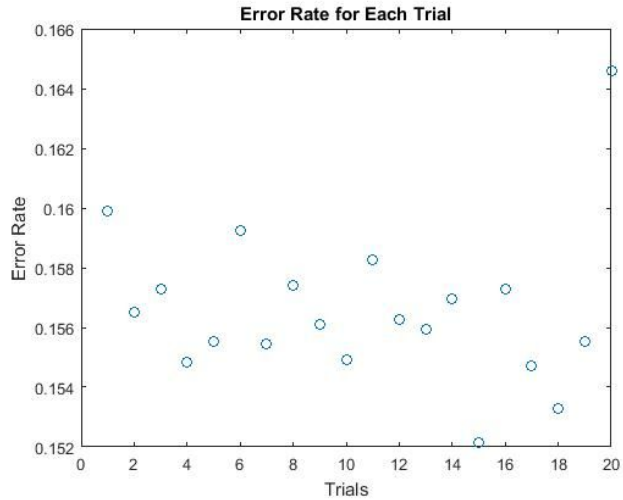

Figure 4. Error rate for each validation. Minimum error rate is found at trial 15.

IV. Computational Results

One Layer

The Training set in this project is cross validated 20 times, figure 4 shows that the error rate for each validation range from 0.152 to 0.165. The weight matrix A with the smallest error rate is chosen and applied in training.

After running the program for 5 times, we found an average overall error rate of 0.1465 with a root mean square error of 0.3536. This indicates that over 85% digits are correctly recognised.

MultiLayer

Matlab's toolbox DNN published by Mr.Tanaka is used to carry out the results for multilayer mnist recognition. In this program, input units are first pretrained, then trained using the Deep Belief Nets (DBN) model. The file used to test the result can be found in the toolbox using this path: '*Deep Neural Network\code\DeepNeuralNetwork\mnist'.*
'evaMNIST.m' provided numerical result for the error rates, rates may change based on the parameters defined in 'trainMNIST.m':

TrainNum: number of images used in the train set;
TestNum: number of images used in the test set;
nodes: the number of units for each layer. length(nodes) - 1 = number of layer;
MaxIter: number of iterations for pretrain and train cycles.
StepRatio: learning step size;
Train rmse, error rate: The mean square error and error rate for train set.
Test rmse, error rate: The mean square error and error rate for test set.

Table 1. trainMNIST.m Paramaters and Result.

Set	TrainNum	TestNum	nodes	MaxIter	StepRatio	Train rmse	error rate	Test rmse	error rate
1	10000	100	[783 300 100 10]	100	0.1	0.084	0.0414	0.086	0.0444
2	10000	100	[783 300 10]	100	0.1	0.0856	0.0387	0.0887	0.0410
3	5000	100	[784 300 10]	100	0.1	0.160	0.1298	0.157	0.1235
4	5000	100	[784 300 10]	100	0.01	0.172	0.1386	0.170	0.1335
5	5000	100	[784 300 10]	100	0.1	0.120	0.0771	0.118	0.0743
6	10000	100	[784 300 10]	100	0.1	0.116	0.0532	0.109	0.0573

Multiple combinations of parameters are tested, the parameter values and results are as shown in table 1. All other parameters are kept the same.

Observation:

1. Set 1 uses 3 layers and only 10,000 images as the train set. The resulting error rate is 0.044, much lower than 1 layer. Set 2 used only 2 layers while other parameters are kept the same as set 1. A small decrease in performance is noticed. Same pattern can be found in set 5 and 6.
2. Step ratio is changed from 0.1 to 0.01 and accuracy dropped 1%.
3. Set 6 uses double times of train num than set 5. Error rate for set 6 is ~2% lower.

V. Summary and Conclusions

In this project, we have implemented a one layer DNN model for handwritten digit recognition. 85% of the images are accurately recognized by this model. We also tested the performance of DNN using the DNN toolbox in Matlab. Our result has evidently shown that:

1). As train set size increases, accuracy increases significantly.

2). As the DNN model becomes deeper, in other words, the number of layers increases, accuracy increases.

Reference

[1] Nielsen, Michael. “Using Neural Nets to Recognise Handwritten Digits” Source:http://neuralnetworksanddeeplearning.com/chap1.html Jan, 2017.

[2] LeCun, Yann; Corinna Cortes; Christopher J.C. Burges. "MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges". Retrieved 16 March 2017.

[3] Tanaka, Masayuki. Deep Neural Networks Mathworks File Exchange. Source:https://www.mathworks.com/matlabcentral/fileexchange/42853-deep-neural-network 29 July, 2013.

[4] Stutz, David. “Introduction to Neural Networks” 10 February , 2014.

[5]UFLFL, “Using the MNIST Dataset”, Available at: http://ufldl.stanford.edu/wiki/resources/mnistHelper.zip. 3 May 2011.

Appendix A MATLAB functions used and brief implementation explanation

randperm - random permutation, used to select random indices.

zeros - create a matrix filled with 0s.

pinv - Moore Penrose pseudoinverse of a matrix, used to calculated the pseudo inverse of a non-square matrix.

max, min - return the maximum/minimum element and its index.

Appendix B MATLAB codes

One Layer.m

```
clear all; close all; clc
% Loading data
trainSet = loadMNISTImages('train-images-idx3-ubyte');
trainLabel = loadMNISTLabels('train-labels-idx1-ubyte');
testSet = loadMNISTImages('t10k-images-idx3-ubyte');
testLabel = loadMNISTLabels('t10k-labels-idx1-ubyte');
[trainDim, trainNum] = size(trainSet);
[testDim, testNum] = size(testSet);
%%
errorRate = [];
for totalRun = 1: 5
validError = [];
ite = 5; % Iterations for validation
trainSize = 50000;
validSize = trainNum - trainSize;
ASet = [];
%% Cross Validation
for count = 1: ite
      % Pick random indices.
      index = randperm(trainNum);
      indexTrain = index(1:trainSize);
      indexValid = index(trainSize+1: end);
      % Seperating training/testing data
      vTrainLabel = zeros(10, trainSize);
      validLabel = zeros(10, validSize);
      vTrainSet = zeros(trainDim,trainSize);
      validSet = zeros(trainDim, validSize);
      for i = 1 : trainSize
            vTrainLabel(trainLabel(indexTrain(i)) + 1,i) = 1;
            vTrainSet(:,i) = trainSet(:, indexTrain(i));
      end

      for j = 1 : validSize
            validLabel(trainLabel(indexValid(j))  + 1,j) = 1;
            validSet(:,j) = trainSet(:, indexValid(j));
      end
      A = vTrainLabel * pinv(vTrainSet);% A = S* pinv(X)
```

```
        vResult = A * validSet;
        [M,I] = max(vResult); % Maximum, index of Maximum

        vResultLabel = zeros(10, validSize);
        for k = 1 : validSize
              vResultLabel(I(k),k) = 1;
        end

        vError = vResultLabel - validLabel;
        error = nnz(vError)/2; % #non-zero = 2 * mismatch.
        validError = [validError; error/validSize];
        ASet = [ASet; A];

end

plot(validError,'o');
title('Error Rate for Each Trial')
xlabel('Trials');
ylabel('Error Rate')

testLabelMatrix = zeros(10, testNum);
resultMatrix = zeros(10, testNum);

%Pick A that gives minimum error rate.
[M, I] = min(validError);
A = ASet((I-1)*10+1:I*10,:);
% Final Result.
result = A*testSet;
for k = 1 : testNum
            [M,I] = max(result);
            resultMatrix(I(k),k) = 1;
end
for i = 1 : testNum
      testLabelMatrix(testLabel(i) + 1,i) = 1; % resulting matrix that has
1 at label_value + 1 position, since value range from 0 to 9
end

errorMatrix = resultMatrix - testLabelMatrix;
errorCount = nnz(errorMatrix)/2;
errorRate = [errorRate; errorCount/testNum];
end
figure;
plot(errorRate, 'o'); %0.1473
title('Error Rate for Test Set');
xlabel('Run #');
ylabel('Error Rate');
```